Oracle Endeca Commerce

Experience Manager Cartridge Developer's Guide

Version 3.1.0 • July 2012

Contents

Copyright and disclaimer

Preface

Oracle Endeca's Web commerce solution enables your company to deliver a personalized, consistent customer buying experience across all channels — online, in-store, mobile, or social. Whenever and wherever customers engage with your business, the Oracle Endeca Web commerce solution delivers, analyzes, and targets just the right content to just the right customer to encourage clicks and drive business results.

Oracle Endeca Commerce is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Oracle Endeca Commerce enables businesses to help guide and influence customers in each step of their search experience. At the core of Oracle Endeca Commerce is the MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Endeca Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. Endeca Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

Oracle Endeca Experience Manager is a single, flexible solution that enables you to create, deliver, and manage content-rich, cross-channel customer experiences. It also enables non-technical business users to deliver targeted, user-centric online experiences in a scalable way — creating always-relevant customer interactions that increase conversion rates and accelerate cross-channel sales. Non-technical users can control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Oracle Endeca Experience Manager, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide describes how to develop custom cartridges. It includes an overview of the Assembler processing model and provides examples demonstrating how to customize Assembler behavior through cartridge handlers.

This guide assumes that you have installed Oracle Endeca Commerce with Experience Manager, and that you have deployed the Discover Electronics reference application as described in the *Tools and Frameworks Installation Guide*. It also assumes that you are familiar with Experience Manager and Assembler concepts as described in the *Assembler Application Developer's Guide*.

The Assembler is implemented in Java, so the examples in this guide are primarily Java-based.

Who should use this guide

This guide is intended for developers using Oracle Endeca Experience Manager who need to customize or extend the Endeca Assembler for a specific application.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Oracle Support

Oracle Support provides registered users with important information regarding Oracle Endeca software, implementation questions, product and solution help, as well as overall news and updates.

You can contact Oracle Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1

Cartridge Basics

This section introduces the basic components of a cartridge by examining how they work together in a "Hello, World" example cartridge.

First steps with a new cartridge

To begin, we'll define a new cartridge and use Endeca Workbench to configure it to display on a page.

To create and configure a basic "Hello, World" cartridge:

1. Create a cartridge template.
   a) Open a new plain text or XML file.
   b) Type or copy the following into the contents of the file:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"

                 xmlns:editors="editors"
                 type="SecondaryContent"
                 id="Hello">
  <Description>A sample cartridge that can display a simple
    message.</Description>
  <ThumbnailUrl>/ifcr/tools/xmgr/img/template_thumbnails/sidebar_con¬
tent.jpg</ThumbnailUrl>
  <ContentItem>
    <Name>Hello cartridge</Name>
    <Property name="message">
      <String/>
    </Property>
    <Property name="messageColor">
      <String/>
    </Property>
  </ContentItem>

  <EditorPanel>
    <BasicContentItemEditor>
      <editors:StringEditor propertyName="message" label="Message"/>
      <editors:StringEditor propertyName="messageColor"
        label="Color"/>
    </BasicContentItemEditor>
  </EditorPanel>
</ContentTemplate>
```

c) Save the file with the name `SecondaryContent-Hello.xml` in the templates directory of your Discover Electronics application, for example: `C:\Endeca\apps\Discover\config\cartridge_templates`.

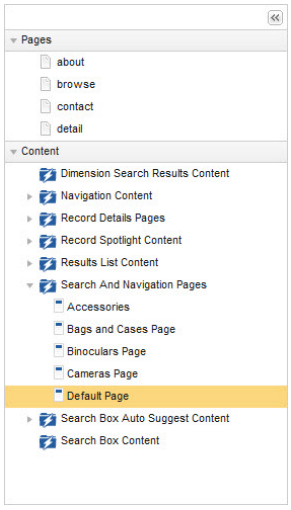
2. Upload the template to Endeca Workbench.
   a) Open a command prompt and navigate to the `control` directory of your deployed application, for example, `C:\Endeca\apps\Discover\control`.
   b) Run the `set_templates` command.

```
C:\Endeca\apps\Discover\control>set_templates.bat
Removing existing cartridge templates for Discover
Setting new cartridge templates for Discover
Finished setting templates

C:\Endeca\apps\Discover\control>
```

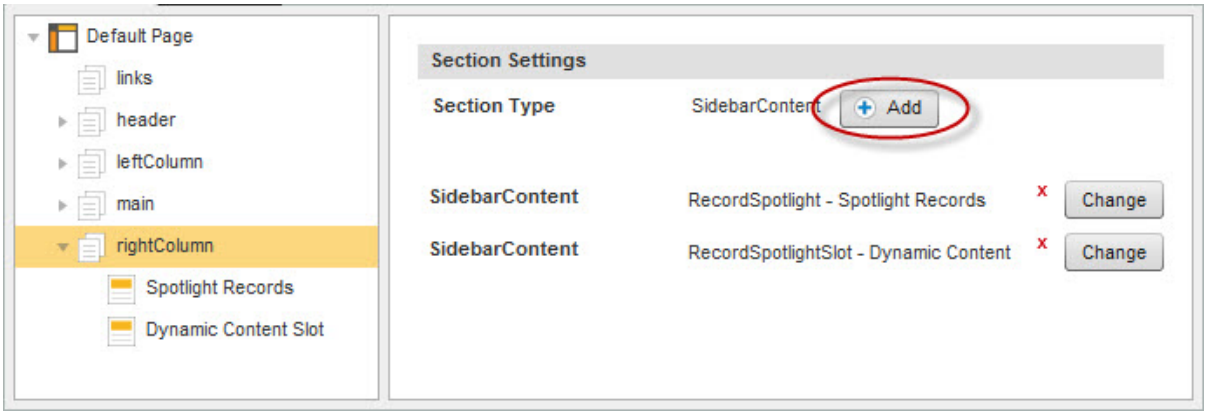
3. Add the cartridge to a page.
   a) Open Endeca Workbench in a Web browser.

      The default URL for Workbench is `http://<workbench-host>:8006`. The default **Username** is `admin` and the default **Password** is `admin`.

   b) From the launch page, select **Experience Manager**.
   c) In the tree on the left, select **Search and Navigation Pages** under the Content section, then select the **Default Page**.

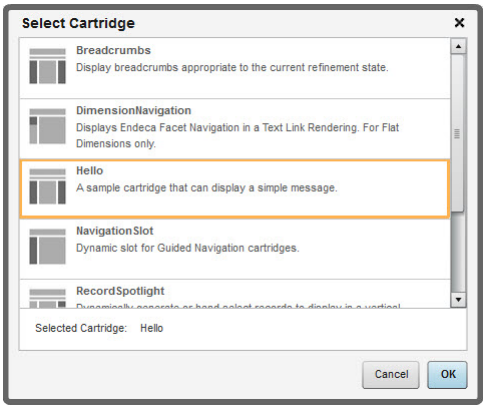


   d) In the Edit Pane on the right, select the right column section from the Content Tree in the bottom left.



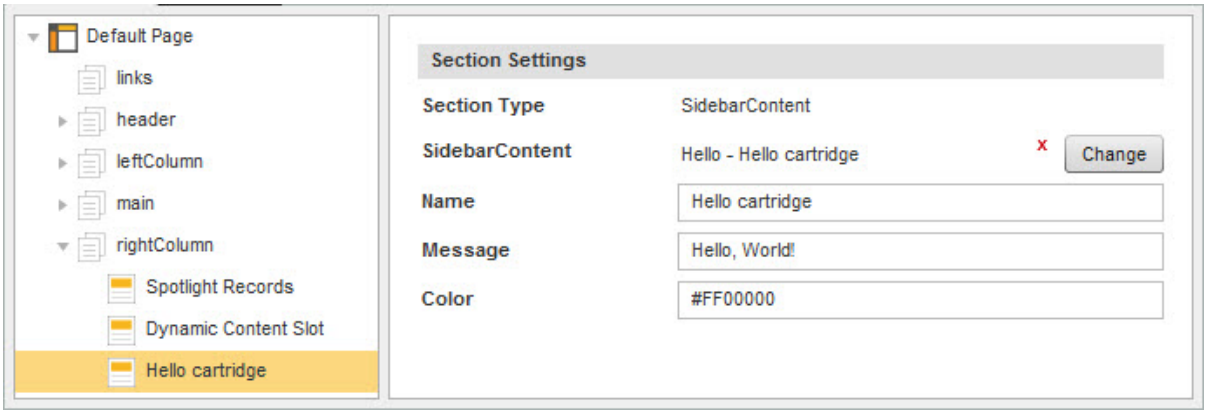
e) Click **Add**.

The cartridge selector dialog displays.



f) Select the **Hello** cartridge and click **OK**.

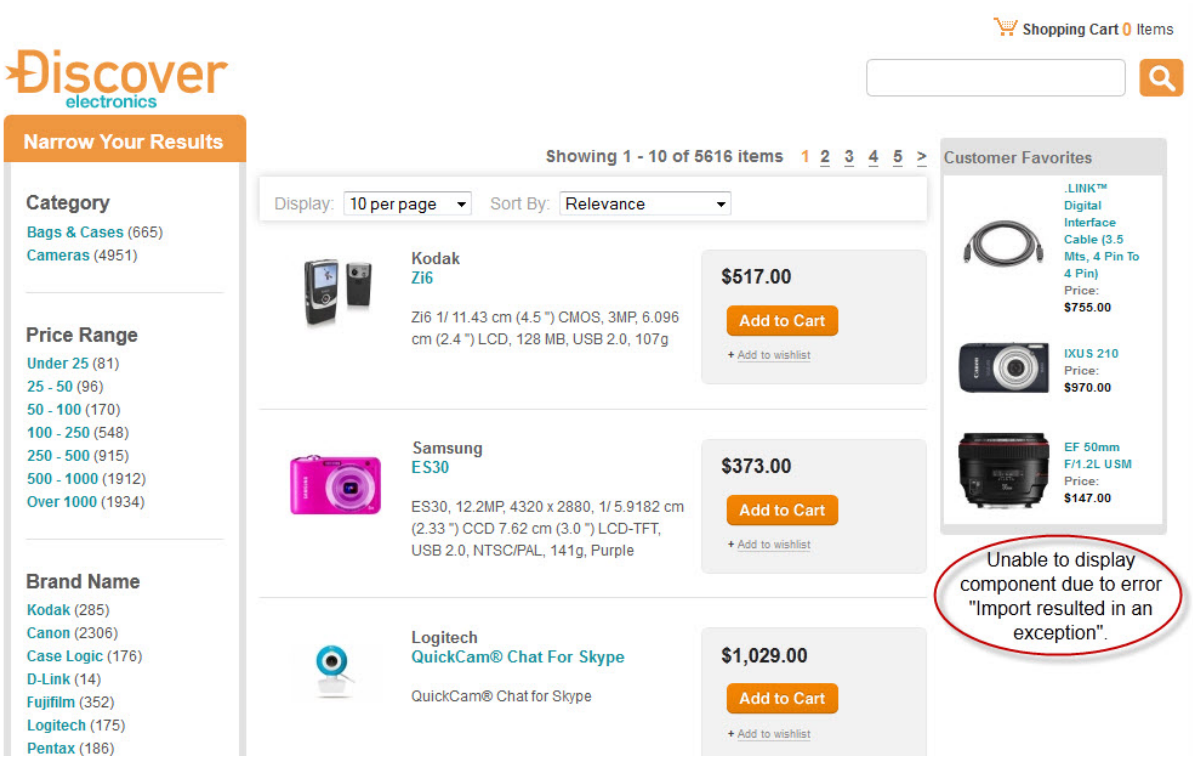
g) Select the new Hello cartridge from the Content Tree on the left and configure it as shown:



h) Click **Save Changes** in the upper right of the page.

4. Try to view the cartridge in the Discover Electronics application.

a) In a Web browser, navigate to `http://<workbench-host>:8006/discover-authoring/`.



The error displays because we have not yet created a renderer for the Hello cartridge.

b) Scroll down to the bottom of the page and click the **json** link view the serialized Assembler response model that respresents the current page.

Oracle recommends that you use a browser or install a plugin that supports native JSON display. Otherwise, you can download the JSON response as a file.

Alternatively, you can click the **xml** link to view the same response in XML. In this guide, we use the JSON format when examining the Assembler response.

The following shows the JSON representation of the page with most of the tree collapsed, highlighting the data for the cartridge that we just added.

```
{
    "@type": "ResultsPageSlot",
    "name": "Browse Page",
    "contentCollection": "Search And Navigation Pages",
    "ruleLimit": "1",
    "contents": [
        {
            "@type": "ThreeColumnNavigationPage",
            "name": "Default Page",
            "title": "Discover Electronics",
            "metaKeywords": "camera cameras electronics",
            "metaDescription": "Endeca eBusiness reference application.",
            "links": [ ],
```

```
            "header": [ … ],
            "leftColumn": [ … ],
            "main": [ … ],
            "rightColumn": [
                { … },
                { … },
                {
                    "@type": "Hello",
                    "name": "Hello cartridge",
                    "message": "Hello, World!",
                    "messageColor": "#FF0000"
                }
            ]
        }
    ],
    …

}
```

In the next section, we'll create a simple renderer that displays the message based on the values configured in Experience Manager.

Adding a basic renderer

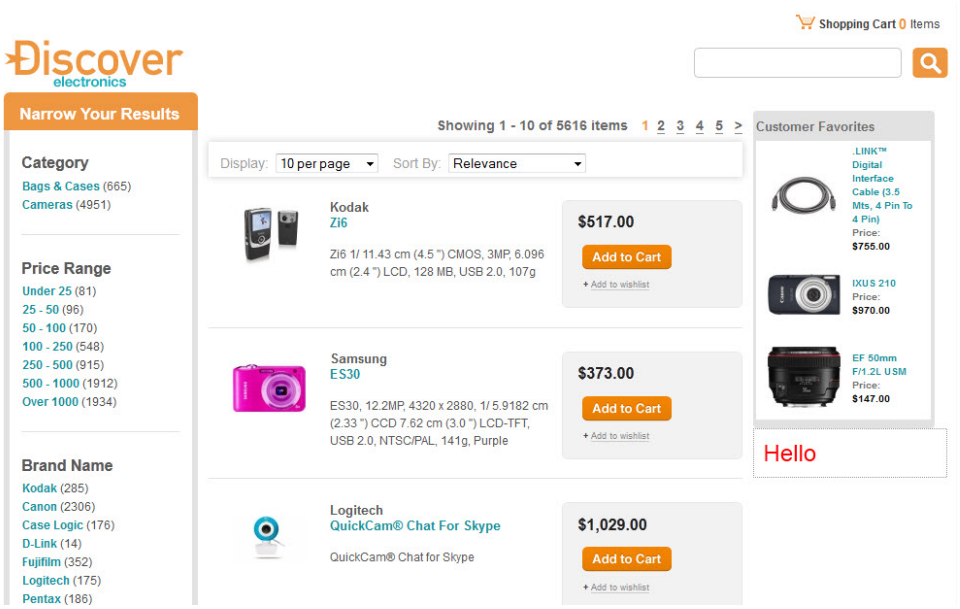
While there is no one way to write rendering code for an application, in this example we'll write a simple JSP renderer for our basic cartridge.

To write a basic "Hello, World" renderer:

1. Create a new JSP page and type or copy the following:

```
<%@page language="java" pageEncoding="UTF-8"
  contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>
<div style="border-style: dotted; border-width: 1px;
  border-color: #999999; padding: 10px 10px">
  <div style="font-size: 150%;
    color: ${component.messageColor}">${component.message}
  </div>
</div>
```

2. Save the renderer to `discover-electronics-authoring/WEB-INF/views/desktop/Hello/Hello.jsp`.
3. Refresh the Discover Electronics authoring application at `http://<workbench-host>:8006/discover-authoring/` to see the result.



Deeper dive on the example cartridge

Now that we have created a basic example cartridge, let's examine each of the cartridge elements more closely.

As we have seen, the high-level workflow for creating a basic cartridge is:

1. Create a cartridge template and upload it to Endeca Workbench.
2. Use Experience Manager to create and configure and instance of the cartridge.
3. Add a renderer to the front-end application.

Step 2 is necessary during development in order to have a cartridge instance with which to test. However, once the cartridge is complete, the business user is typically responsible for creating and maintaining cartridge instances in Experience Manager.

In the following sections, we'll describe each of these elements of the cartridge in greater detail.

The cartridge template

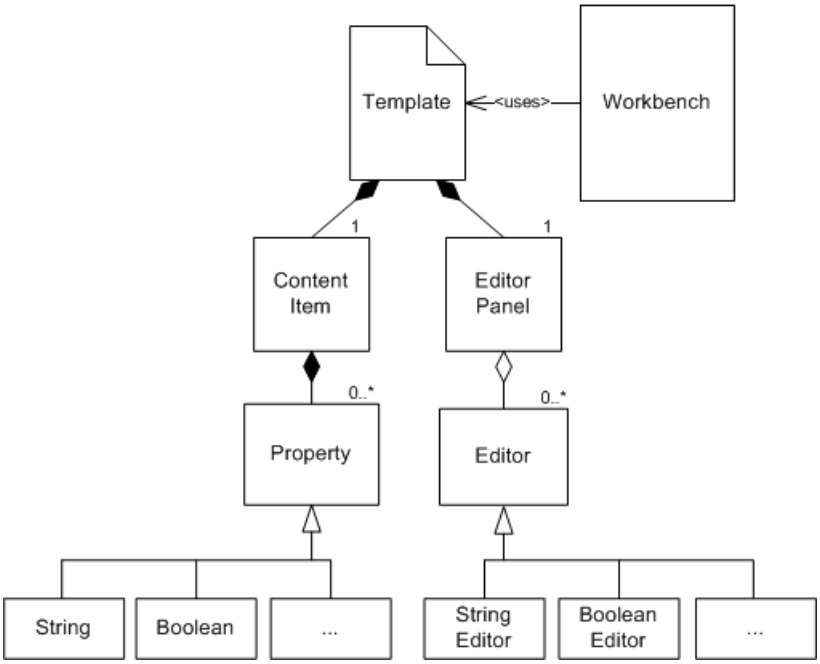
The template defines the configuration that the business user can specify in Endeca Workbench using Experience Manager.

The template contains two main sections: the `<ContentItem>` element and the `<EditorPanel>` element.

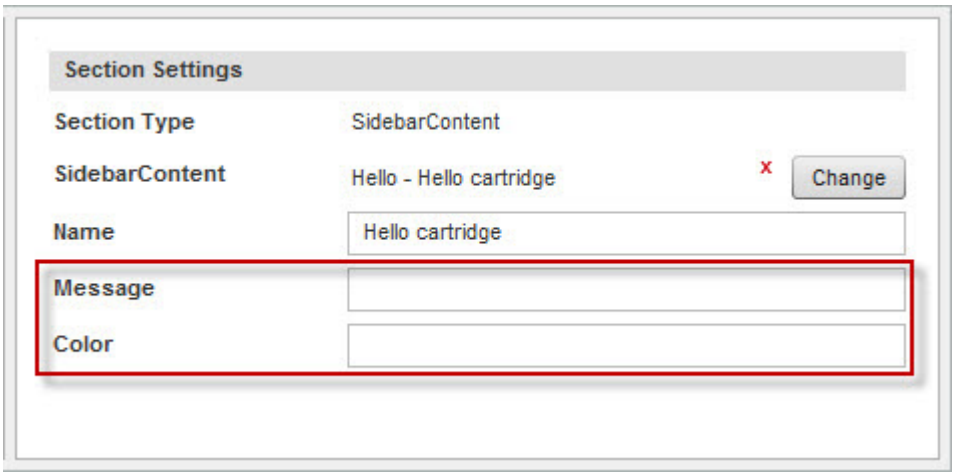
The *content item* is a core concept in Assembler applications that can represent both the configuration model for a cartridge and the response model that the Assembler returns to the client application. A content item is a map of properties, or key-value pairs. The `<ContentItem>` element in the template defines the prototypical content item and its properties, similar to a class or type definition.

The `<EditorPanel>` defines the interface that can be used in Experience Manager to configure the properties of the content item. The *editor panel* is composed of a number of *editors*. The editors provide

the UI controls that the business user can use to specify the property values for a particular instance of that cartridge.



In our example template, we defined two string properties named `message` and `messageColor` and attached two simple string editors to those properties. The result looks like this in Experience Manager:



For more information about creating and managing cartridge templates, refer to the *Assembler Application Developer's Guide*.

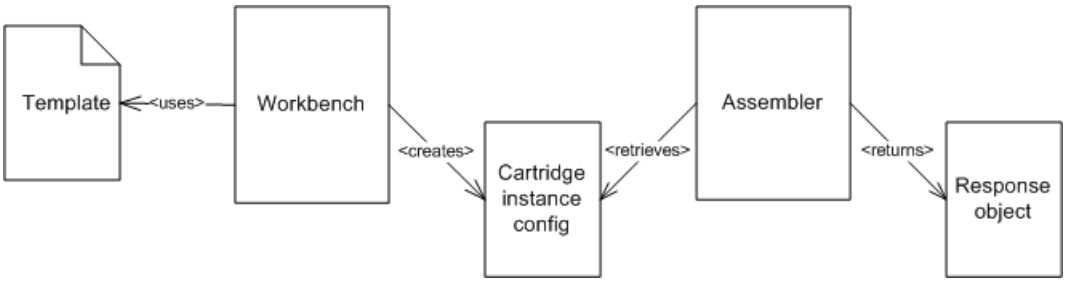
The cartridge instance configuration

The business user creates and configures instances of cartridges in Experience Manager based on a template. During cartridge development you need to create at least one instance of a cartridge for testing.

Experience Manager writes this cartridge instance configuration as XML. You can view the XML representation of the configuration using the **XML View** tab in Experience Manager. The following shows the XML that corresponds to the configured instance of our example cartridge:

Note the similarities to the `<ContentItem>` portion of the template that we created. At this stage, the values of the string properties have been filled in based in the input in the **Content Editor** pane.

The Assembler retrieves this configuration at runtime and uses it to build the response model that it returns to the client application.



For any given cartridge, the default behavior is for the Assembler to do no processing on the configuration and simply return the configuration content item as a map of properties. That is, the response object is the same as the configuration object unless specific processing logic is defined in the Assembler for that cartridge.

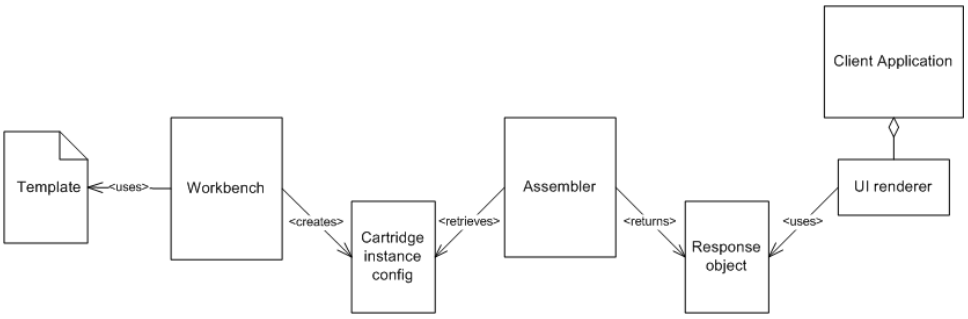
The cartridge renderer

As a best practice, the client application should be composed of modular rendering components, each corresponding to a particular cartridge.

Recall the contents of the Assembler response object corresponding to the example cartridge:

```
{
    "@type": "Hello",
    "name": "Hello cartridge",
    "message": "Hello, World!",
    "messageColor": "#FF0000"
}
```

For each cartridge, the `@type` of the response object corresponds to the `id` of the template that was used to create it. The Discover Electronics application uses this type to identify the appropriate renderer to use for this content item.

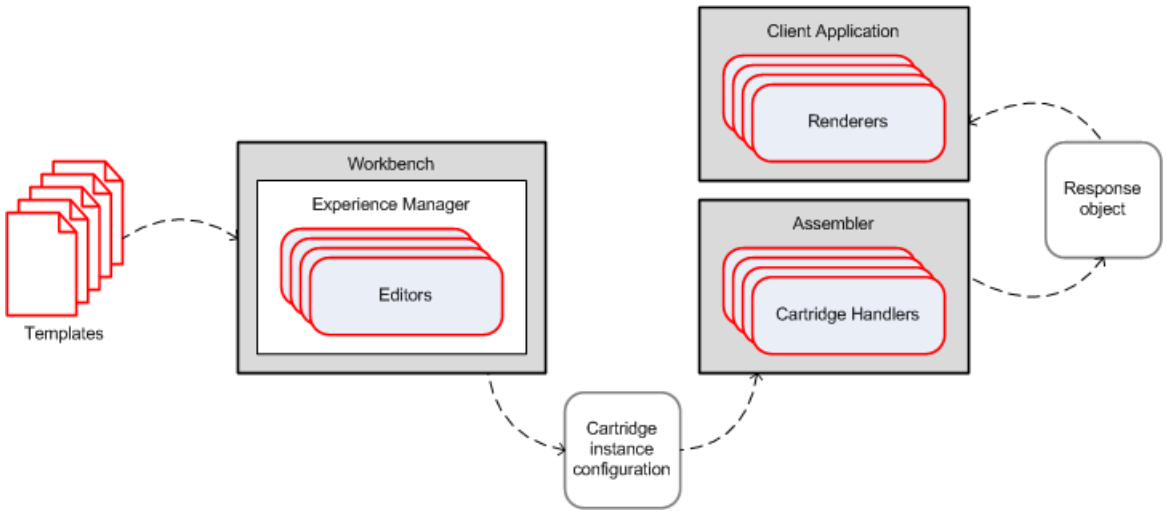


The logic for mapping response objects to the appropriate renderer is contained in `include.tag` in the reference application.

Overview of cartridge extension points

Cartridges are made up of several components that may be customized for specific purposes.

The following diagram shows the parts of a cartridge and where they fit within the overall architecture:



The cartridge *template* defines the configuration options that are available to the business user in Workbench. The Experience Manager interface is composed of *editors*, or Flex components that provide UI controls for specifying property values. Experience Manager produces the *cartridge instance configuration* that is consumed by the Assembler. During the processing of a query, the Assembler may invoke *cartridge handlers* that define specific processing logic for particular cartridges. Using these cartridge handlers, the Assembler produces the *response object* that it returns to the client application. Typically, the client application includes modular *renderers* that are intended to handle a particular cartridge.

We created a basic template and renderer in the example cartridge. We also inspected the cartridge instance configuration generated in Workbench and the response returned by the Assembler. In the example cartridge, both the configuration and the response model were generic content items that are simple maps of properties. Many of the core Endeca cartridges have strongly typed configuration models and response objects associated with them that extend from the basic content item. This makes it easier to understand the expected input to and output from the core cartridge handlers, and also enables reuse of the models for the core cartridges. Strongly typed configuration beans also make

it possible to configure default values for cartridge properties via Spring. Creating strongly typed model objects for the Assembler configuration and response is not required when developing cartridges.

In the following sections, we discuss how to customize the Experience Manager interface using editors, and how to define custom processing logic in the Assembler using cartridge handlers.

Chapter 2

Customizing the Experience Manager interface

Experience Manager provides a set of standard editors that you can use in cartridge templates as well as the ability to develop custom editors.

Adding embedded user assistance to a cartridge

You can provide embedded assistance for the business user in the Experience Manager interface by specifying it in the cartridge template.

In our example cartridge, we provided two simple text fields for the business user to enter a message and the desired color. This user interface makes it unclear what values are allowed or expected for those fields. The template schema for configuring editors allows you to supply a short descriptive label for each field, but sometimes additional context can be helpful. For such cases, you can use the `bottomLabel` attribute to provide further information.

To add additional guidance for the business user to the example cartridge:

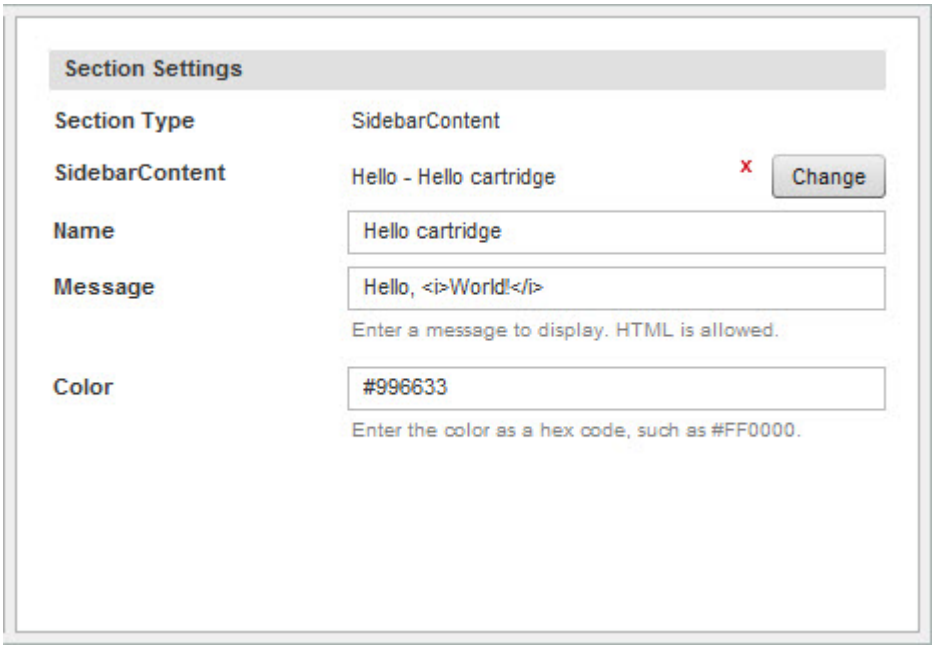
1. Open the template file (`SecondaryContent-Hello.xml`) that you previously created.
2. Add a `bottomLabel` attribute to each editor in the `<EditorPanel>`, as in the example below:

```
<EditorPanel>
  <BasicContentItemEditor>
    <editors:StringEditor propertyName="message" label="Message"
      bottomLabel="Enter a message to display. HTML is allowed."/>
    <editors:StringEditor propertyName="messageColor"
      label="Color" bottomLabel="Enter the color as a hex code, such
      as #FF0000."/>
  </BasicContentItemEditor>
```

   This additional label text can be configured for all editors built using the Experience Manager SDK, including all the standard editors. For the full content of the updated template, see the example below.

3. Save and close the template.
4. Upload the template by running the `set_templates` script.

The resulting user interface in Experience Manager looks like the following:



The following shows the complete content of the updated template:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
                 xmlns:editors="editors"
                 type="SecondaryContent"
                 id="Hello">
  <Description>A sample cartridge that can display a simple
    message.</Description>
  <ThumbnailUrl>/ifcr/tools/xmgr/img/template_thumbnails/sidebar_con¬
tent.jpg</ThumbnailUrl>
  <ContentItem>
    <Name>Hello cartridge</Name>
    <Property name="message">
      <String/>
    </Property>
    <Property name="messageColor">
      <String/>
    </Property>
  </ContentItem>
  <EditorPanel>
    <BasicContentItemEditor>
      <editors:StringEditor propertyName="message" label="Message"
        bottomLabel="Enter a message to display. HTML is allowed."/>
      <editors:StringEditor propertyName="messageColor"
        label="Color" bottomLabel="Enter the color as a hex code, such as
 #FF0000."/>
    </BasicContentItemEditor>
  </EditorPanel>
</ContentTemplate>
```

For more information about label options for Experience Manager editors, refer to the *Assembler Application Developer's Guide*.

Using the core Experience Manager editors

Experience Manager provides a set of editors that can configure primitive property types as well as Endeca-specific features. You specify which editor to use to configure which properties in the `<EditorPanel>` portion of the template.

Even with additional user assistance text, asking the business user to type a hex code into a text field does not provide a very user-friendly experience. One of the standard editors included with Experience Manager is a combo box that can be used to specify a set of valid values for a string property. In this example, we provide a set of colors from which the business user can choose. This not only relieves the business user from typing in a hex code, but it can also ensure that the selected color matches the site's color scheme.

To update the example cartridge to use a combo box editor:

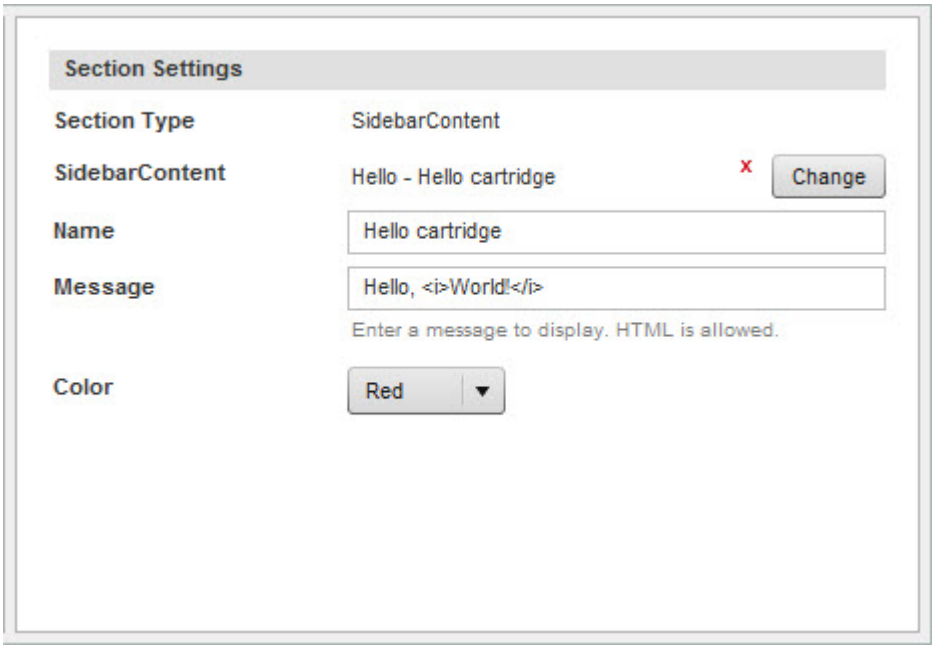
1. Open the template file (`SecondaryContent-Hello.xml`) that you previously created.
2. Replace the string editor configuration for the `messageColor` property with the following:

```
<EditorPanel>
  <BasicContentItemEditor>
    <editors:StringEditor propertyName="message" label="Message"
      bottomLabel="Enter a message to display. HTML is allowed."/>
    <editors:ChoiceEditor propertyName="messageColor" label="Color">
      <choice label="Red" value="#FF0000"/>
      <choice label="Green" value="#00FF00"/>
      <choice label="Blue" value="#0000FF"/>
    </editors:ChoiceEditor>
  </BasicContentItemEditor>
</EditorPanel>
```

   For the full content of the updated template, see the example below.

3. Upload the template by running the `set_templates` script.

The resulting user interface in Experience Manager looks like the following:



Depending on the option that the business user selects, the value of the property is set to the appropriate hex code. You can change the value and refresh the application to see the change.

The following shows the complete content of the updated template:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
                 xmlns:editors="editors"
                 type="SecondaryContent"
                 id="Hello">
<Description>A sample cartridge that can display a simple
  message.</Description>
<ThumbnailUrl>/ifcr/tools/xmgr/img/template_thumbnails/sidebar_con¬
tent.jpg</ThumbnailUrl>
<ContentItem>
    <Name>Hello cartridge</Name>
    <Property name="message">
        <String/>
    </Property>
    <Property name="messageColor">
        <String/>
    </Property>
</ContentItem>
<EditorPanel>
    <BasicContentItemEditor>
      <editors:StringEditor propertyName="message" label="Message"
        bottomLabel="Enter a message to display. HTML is allowed."/>
        <editors:ChoiceEditor propertyName="messageColor" label="Color">
            <choice label="Red" value="#FF0000"/>
            <choice label="Green" value="#00FF00"/>
            <choice label="Blue" value="#0000FF"/>
        </editors:ChoiceEditor>
    </BasicContentItemEditor>
</EditorPanel>
</ContentTemplate>
```

For more information about the standard Experience Manager editors and their configuration, refer to the *Assembler Application Developer's Guide*.

About custom editors

If none of the standard editors meet your needs, you can develop your own editors using the Experience Manager Editor SDK.

You may want to develop an editor if:

- You want to allow the business user to configure more advanced properties such as lists or maps of key-value pairs.
- You want to provide a more advanced interface for the business user, such as a list that enables drag-and-drop.
- You want the editor options to be populated dynamically from an external system rather than configured in the template.
- You want the behavior of one editor or UI control to be linked to the state of another.

For more information about the Experience Manager Editor SDK and developing Experience Manager editors, refer to the *Experience Manager Editor Developer's Guide*.

Chapter 3

About Cartridge Handlers and the Assembler

This section provides an overview of the Endeca Assembler. It describes the Assembler processing model and core interfaces as well as how to implement a cartridge handler.

Overview of the Assembler processing model

The core of the Assembler is the `assemble()` method, which takes a content item representing a cartridge instance configuration and invokes cartridge handlers to process the configuration into the response content item.

The Assembler uses the visitor pattern to traverse the configuration content item and any child content items, invoking the appropriate cartridge handler, if any, for each content item. (Recall that a content item can have properties that are themselves content items or lists of content items.)

The Assembler makes two passes over the content tree:

1. In the first pass, the Assembler calls `CartridgeHandler.initialize()` followed by `Cartridge¬Handler.preprocess()` on each cartridge in the tree. This is a pre-order traversal of the tree (working from the top of the tree down through its children), so cartridge handlers may add or modify child content items at this stage.
2. In the second pass, the Assembler calls `CartridgeHandler.process()` on each cartridge, which returns the response content item for that cartridge. This is a post-order traversal of the tree (working from the bottom up), so all child content items are processed before the parent. The response object for the root content item of the tree contains the response objects for all its child cartridges.

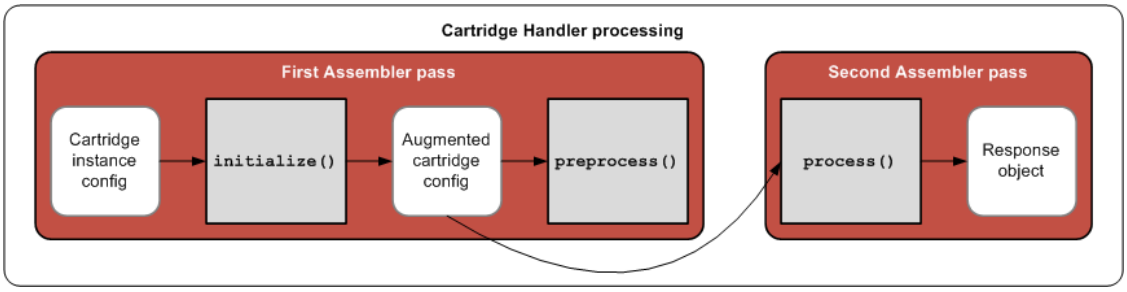
The default implementation of the Assembler uses Spring to map each cartridge to the appropriate handler based on its cartridge type, which corresponds to the `id` of the template that was used to create the cartridge instance. If no cartridge handler is defined for a particular cartridge type, the instance configuration is passed through as the response model.

The Assembler is typically invoked with a `ContentInclude` or `ContentSlot` item. The corresponding handlers provide two alternatives for retrieving the relevant cartridge instance configuration for a particular request, based on either a URI or a set of MDEX Engine trigger criteria. For more information about invoking the Assembler, refer to the *Assembler Application Developer's Guide*.

About the CartridgeHandler interface

A cartridge handler takes a content item representing the cartridge instance configuration as input and is responsible for returning the response as a content item.

The `CartridgeHandler` interface defines three methods: `initialize()`, `preprocess()`, and `process()`.



The `initialize()` method provides an opportunity for the cartridge handler to augment the cartridge instance configuration specified in Experience Manager with configuration from other sources. This can be used to define default behavior for a cartridge in the case where there is no Experience Manager configuration, or to override the Experience Manager configuration for the current query. The `initial¬ize()` method should return a content item containing the complete configuration for the cartridge from all possible configuration sources. This augmented configuration item can either be the mutated input content item or a new instance of `ContentItem`, and is used as input to both the `preprocess()` and `process()` methods.

Because the `preprocess()` method is called on all cartridges before `process()` is called on any cartridges, it provides an opportunity to coordinate processing between cartridges. Many of the core Endeca cartridges make use of this mechanism in order to consoldiate queries to an MDEX Engine among several cartridges during the course of a single assembly cycle.

The `process()` method is responsible for returning a `ContentItem` that represents the cartridge response.

A cartridge handler need not define any behavior for `initialize()` or `preprocess()`. The `Ab¬stractCartridgeHandler` class exists to simplify the task of implementing the `CartridgeHandler` interface. It provides empty implementations for `initialize()` and `preprocess()`. Subclasses of `AbstractCartridgeHandler` need only implement the `process()` method to return the response object. They can optionally override the `initialize()` and `preprocess()` methods.

About initializing the cartridge configuration

The `initialize()` phase in the cartridge processing life cycle enables the cartridge handler to synthesize the complete configuration for the cartridge from several sources.

The configuration content item that is passed in to the assembly process is the cartridge instance configuration from Experience Manager, however, any given cartridge may also have other configuration sources.

In a typical scenario, a cartridge has some default behavior that can be specified as a property value in a Spring context file. A business user can specify a value for a specific instance of a cartridge using Experience Manager. The site visitor may also have the ability to override either the default or the cartridge instance setting from the client application. For example, in the Results List cartridge, the default value for records per page is 10. The business user can set this value to 25 in Experience

Manager, and the site visitor can choose to display 50 records by selecting the appropriate option on the site.

The Assembler API includes the `ConfigInitializer` utility class with the method `initialize()`. The default implementation of `initialize()` layers the cartridge configuration in the following order (from lowest to highest):

1. **Default configuration**, typically defined in the Spring configuration for the cartridge handler
2. **Cartridge instance configuration**, typically created in Experience Manager and passed in as the configuration content item
3. **Request-based configuration** parsed from the HTTP request parameters, using the `Request¬ ParamMarshaller` helper class

The `ConfigInitializer` class also provides methods for additional layering of configuration. Subclasses can override `ConfigInitializer` to define custom layering behavior, for example, to incorporate configuration saved in the session state.

About the NavigationCartridgeHandler class

The core Endeca cartridges that make queries to an MDEX Engine use cartridge handlers that extend from `NavigationCartridgeHandler`.

The `NavigationCartridgeHandler` makes use of the two-pass Assembler processing model to consolidate MDEX Engine queries across cartridges.

In the `preprocess()` phase, the cartridge handler calls `createMdexRequest()` but does not execute the request. In subsequent calls to `createMdexRequest()` by other handlers, the MDEX resource broker determines whether the new request can be consolidated with an existing request in order to minimize the number of queries to the MDEX Engine for a single assembly cycle.

During the `process()` phase, the handler calls `executeMdexRequest()` to retrieve the results. The actual query to the MDEX Engine is executed when the first handler in the assembly cycle calls `executeMdexRequest()` and the results are cached for all subsequent handlers that try to execute the same request.

You can use a similar approach if you have multiple cartridges that need to make requests to the same external resource and can achieve efficiencies by consolidating requests across cartridges.

For further information about the `NavigationCartridgeHandler` class, refer to the *Endeca Assembler API Reference* (Javadoc).

Implementing a cartridge handler

You add a cartridge handler by writing a Java class that implements the `CartridgeHandler` interface and configuring the Assembler to use the new handler in the Spring context file.

In this example, we update our "Hello, World" cartridge to do some simple string manipulation on the message that was specified in Experience Manager. Because this cartridge does not use any configuration other than the cartridge instance configuration from Experience Manager and does not need to do any preprocessing, we can extend `AbstractCartridgeHandler`.

To add a cartridge handler to the example cartridge:

1. Create a new Java class in the package `com.endeca.sample.cartridges` and type or copy the following:

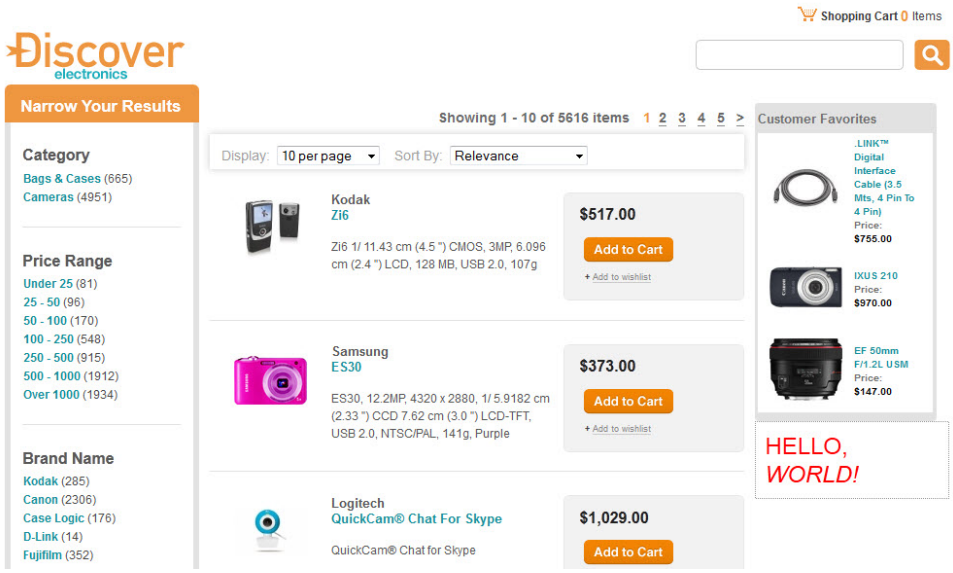
```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;

public class UppercaseCartridgeHandler extends AbstractCartridgeHandler

{
  //=========================================================================

   // The cartridge handler 'process' method
  public ContentItem process(ContentItem pContentItem) throws Cartridge¬
HandlerException
   {
     // Get the message property off of the content item.
     final String message = (String) pContentItem.get("message");
     // If the message is non-null, uppercase it.
     if (null != message) {
        pContentItem.put("message", message.toUpperCase());
     }
     return pContentItem;
   }
}
```

2. Compile the cartridge handler and add the compiled class to your application, for example, by saving it in `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF\classes`.
3. Configure the Assembler to use the `UppercaseCartridgeHandler` for the Hello cartridge.
   a) Navigate to the `WEB-INF` directory of your application, for example, `%ENDECA_TOOLS_ROOT%\reference\discover-electronics-authoring\WEB-INF`.
   b) Open the `assembler-context.xml` file.
   c) Add the following in the `CARTRIDGE HANDLERS` section:

```
    <!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ~ BEAN: CartridgeHandler_Hello
    -->
    <bean id="CartridgeHandler_Hello"
    class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
    scope="prototype" />
```

   d) Save and close the file.
4. Restart the Endeca Tools Service.
5. Refresh the authoring instance of the application.

The message now displays in all-uppercase letters.



Cartridge handler development scenarios

You should write a cartridge handler in cases where you need to perform some processing on the cartridge instance configuration before sending the response to the client application.

It is always possible to do processing in the client application, but encapsulating the business logic in an extension to the Assembler provides several advantages:

- It makes the rendering code cleaner and easier to maintain.
- It centralizes the processing in one place so that the results can be consumed by multiple client applications including across multiple channels such as desktop, mobile, and others.
- It provides an opportunity for coordinating processing across multiple cartridges before returning the response to the client application.

Depending on what the cartridge handler needs to accomplish, your implementation approach may vary. Cartridge handlers must always implement the `process()` method to return the response model.

Scenario	Implementation approach	Example cartridge
Update properties from the cartridge instance configuration in place (data cleansing or manipulation scenario)	Extend `AbstractCartridgeHandler` and override `process()` to update the property values in the input content item	"Hello, World" with UppercaseCartridgeHandler
Use information from the cartridge instance configuration to query an external resource for the information to display	Extend `AbstractCartridgeHandler` and override `process()` to query the resource and insert the results in the output content item	RSS Feed cartridge
Query an external resource, consolidating queries between cartridges within a single assembly cycle for improved performance	Take advantage of the two-pass assembly model with `preprocess()` and `process()` and implement a resource broker that can consolidate queries and manage their execution	NavigationCartridgeHandler

Scenario	Implementation approach	Example cartridge
Augment the results from a core Endeca cartridge with additional information from a non-MDEX resource	Extend the core cartridge and override `process()` to query the resource and add additional properties to the MDEX query results before returning the response	Custom Record Details with availability information
Customize a core Endeca cartridge to modify the MDEX Engine query parameters	Extend the core cartridge and override either `initialize()` or `prepro¬cess()` to modify the query before it is executed	Custom Results List with recommendations
Combine multiple sources of cartridge configuration before processing results	Extend `AbstractCartridgeHandler` or implement the `CartridgeHandler` interface and override `initialize()`, making use of the `ConfigInitializ¬er` and `RequestParamMarshaller` helper classes to generate the complete configuration model	"Hello, World" with layered color configuration

Chapter 4

Sample Cartridges

This section contains sample cartridge customizations that demonstrate how to use the various cartridge extension mechanisms to address different use cases.

About using the sample cartridges

The sample cartridges are intended to demonstrate the cartridge extension mechanisms and provide a model for your own cartridge customizations.

The sample code provided is written to be generic and easy to follow, rather than production-quality code. Oracle recommends that you follow a few best practices when working with the examples:

- Set up a new instance of the Discover Electronics application to use as a sandbox for deploying the sample cartridges. This isolates the samples from the out-of-the-box configuration for Discover Electronics as well as your own application.
- Within your sandbox application, create a separate Spring context file for the custom cartridge handlers described in this guide.
- When copying and pasting examples from this guide, pay attention to the end-of-line marker (¬) that indicates that a long line of text has been wrapped. Ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

The steps described for creating and deploying the components of the sample cartridges correspond to the steps described in previous sections for the "Hello, World" cartridge. If you need additional information to complete a particular step in deploying one of the sample cartridges, refer to the more detailed procedures for the "Hello, World" example.

Setting up a test application based on Discover Electronics

Oracle recommends that you use a test application to test the sample cartridges instead of deploying them in Discover Electronics or your own application.

Because a test application is for development use only, we do not need to deploy a live instance of the application.

To deploy a copy of Discover Electronics to use as a test for the sample cartridges:

1. Deploy a new test application using the Deployment Template.
   a) From a command prompt, navigate to `%ENDECA_TOOLS_ROOT%\deployment_template\bin` (on Windows) or `$ENDECA_TOOLS_ROOT/deployment_template/bin` (on UNIX).

b) Run the deploy script:
   - On Windows: `deploy.bat --app ..\..\reference\discover-data\deploy.xml`
   - On UNIX: `deploy.sh --app ../../reference/discover-data/deploy.xml`

c) Specify the application name `Test` and specify the following ports when prompted:

Port	Recommended value
Live Dgraph	15100
Authoring Dgraph	15102
LogServer	15110

2. Provision the test application.
   a) Ensure that the Endeca HTTP Service and Endeca Tools Service are running.
   b) From a command prompt, navigate to `<APP-DIR>\control` (on Windows) or `<APP-DIR>/control` (on UNIX).
   c) Run `initialize_services`.
   d) Run `load_baseline_test_data`.
   e) Run `baseline_update`.
3. Deploy a copy of the authoring instance of the Discover Electronics application.
   a) Navigate to `%ENDECA_TOOLS_ROOT%\reference` (on Windows) or `$ENDECA_TOOLS_ROOT/reference` (on UNIX).
   b) Make a copy of the directory `discover-electronics-authoring` and save the copy with the name `sandbox` in the same parent directory.
   c) Navigate to the `test` directory and then to the `WEB-INF` subdirectory.
   d) Open `assembler-context.xml` in a text editor.
   e) Locate the `CARTRIDGE SUPPORT` section:

```
<!--

########################################################################

    # CARTRIDGE SUPPORT
    #
    # The following section configures managers and other supporting
objects.
    #
-->
```

   f) In the `mdexResource` bean, update the Dgraph port:

```
<bean id="mdexResource" scope="request"
class="com.endeca.infront.navigation.model.MdexResource">
    <property name="host" value="localhost" />
    <property name="port" value="15102" />
    <property name="recordSpecName" value="common.id" />
</bean>
```

   g) Locate the `Content Sources` section:

```
<!--
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~ Content Sources
-->
```

h) In the `authoringContentSource` bean, update the application name:

```
<bean id="authoringContentSource"" class="com.endeca.infront.con¬
tent.source.AuthoringContentSource"
scope="singleton" lazy-init="true">
    <property name="sitePath" value="/sites/Test"/>
    <property name="rootUrl" value="/ifcr"/>
    <property name="host" value="localhost"/>
    <property name="port" value="8006"/>
    <property name="serviceUrl" value="/ifcr/system/endeca/contentRe¬
solver"/>
    <property name="user" value="admin"/>
    <property name="password" value="admin"/>
</bean>
```

i) In the `authoringMediaSources` bean, update the application name:

```
<bean id="authoringMediaSources" class="java.util.ArrayList" lazy-
init="true">
    <constructor-arg>
        <list>
          <bean class="com.endeca.infront.cartridge.model.MediaSource¬
Config">
                <property name="sourceName" value="IFCRSource" />
                <property name="sourceValue" value="http://local¬
host:8006/ifcr/sites/Test/media/" />
           </bean>
          <bean class="com.endeca.infront.cartridge.model.MediaSource¬
Config">
                <property name="sourceName" value="default" />
                <property name="sourceValue" value="http://local¬
host:8006/ifcr/sites/Test/media/" />
           </bean>
        </list>
    </constructor-arg>
</bean>
```

j) Save and close the file.

k) Navigate to `%ENDECA_TOOLS_CONF%\conf\Standalone\localhost` (on Windows) or `$ENDECA_TOOLS_CONF/conf/Standalone/localhost` (on UNIX).

l) Make a copy of `discover-authoring.xml` and save the copy with the name `test` in the same directory.

m) Open `test.xml` in a text editor.

n) Change the value of `docBase` as follows:

```
docBase="${catalina.base}/../../reference/test"
```

o) Restart the Endeca Tools Service.

4. Validate your new sandbox application:
   a) Navigate to `http://<WorkbenchHost>:8006/login` and verify that `Test` displays as an option in the **Application** drop-down.
   b) Select the `Test` application and verify that the sample page content from Discover Electronics is available in Experience Manager.
   c) In a separate browser window, navigate to the newly deployed sandbox application, at `http://<WorkbenchHost>:8006/test` and verify that it displays.
5. Optionally, update the Workbench configuration to use the test Web application for preview.
   a) Ensure that you are logged in to the Test application in Workbench.

b) Select **Application Configuration**.
c) Specify the URL to the sandbox application (for example, `http://<WorkbenchHost>:8006/test`) as the **Preview URL**.
d) Preview a page from Experience Manager by selecting a page or content item and clicking **Preview** in the upper right.

Creating a Spring context file for sample cartridges

Oracle recommends that you specify the configuration for the sample cartridges in a separate Spring context file from the core Endeca cartridges.

To create a Spring context file for the sample cartridges:

1. Navigate to `%ENDECA_TOOLS_ROOT%\reference\sandbox\WEB-INF` (on Windows) or `$ENDECA_TOOLS_ROOT/reference/sandbox/WEB-INF` (on UNIX).
2. Open `assembler-context.xml` in a text editor.
3. At the top of the file, add the following `import`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
">
   <bean class="org.springframework.beans.factory.config.PropertyPlace¬
holderConfigurer">
        <property name="locations">
            <list>
                <value>WEB-INF/assembler.properties</value>
            </list>
        </property>
    </bean>

    <import resource="endeca-url-config.xml"/>
    <import resource="perf-logging-config.xml"/>
  <import resource="sample-cartridge-config.xml" />
```

4. Delete the configuration for the "Hello, World" sample cartridge that we added in an earlier example.

```
    <!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~ BEAN: CartridgeHandler_Hello
    -->
    <bean id="CartridgeHandler_Hello"
    class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
    scope="prototype" />
```

5. Save and close the file.
6. Create a new file named `sample-cartridge-config.xml` in the same directory with the following contents:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ~ BEAN: CartridgeHandler_Hello
    -->
    <bean id="CartridgeHandler_Hello"
    class="com.endeca.sample.cartridges.UppercaseCartridgeHandler"
    scope="prototype" />

</beans>
```

7. Save and close the file.
8. Validate the new configuration by adding the "Hello, World" cartridge to your new sandbox application.
   a) Copy the "Hello, World" template (`SecondaryContent-Hello.xml`) from the Discover Electronics application (`<APP-DIR>\config\page_builder_templates`) to the sandbox application.
   b) Upload the template to Workbench using the `set_templates` script.
   c) Using Experience Manager, add the cartridge to the default page of the sandbox application and save your changes.
   d) Verify that the `Hello.jsp` renderer and `UppercaseCartridgeHandler` are present in the sandbox Web application. (These should have been included when you copied the Discover Electronics authoring application.)
   e) Refresh the sandbox application (`http://<WorkbenchHost>:8006/sandbox`) and verify that the text you entered in Experience Manager displays, and has been converted to all-uppercase letters.

RSS Feed cartridge

In this example, we build a cartridge that displays items from an RSS feed.

This cartridge enables a business user to specify some basic information about an existing RSS feed in Experience Manager. The cartridge handler fetches the RSS results and returns an output model to the client suitable for rendering.

It demonstrates the following use cases:

- Using a cartridge handler to fetch information from a source other than an MDEX Engine.
- Using the business user configuration from Experience Manager as input into the assembly process and returning a different output model from the configuration model.

In this cartridge, we create the following components:

Component	Description
cartridge template	Enables the business user to specify the URL to an RSS feed and the number of entries to display.
cartridge handler	Fetches results from the RSS feed and returns a number of entries up to the value specified by the business user or the number of entries in the feed, whichever is lower.
cartridge renderer	Displays the name of the feed with a link to the channel URL, and the title and description of each entry with a link to the entry on the original site.

Creating the cartridge template

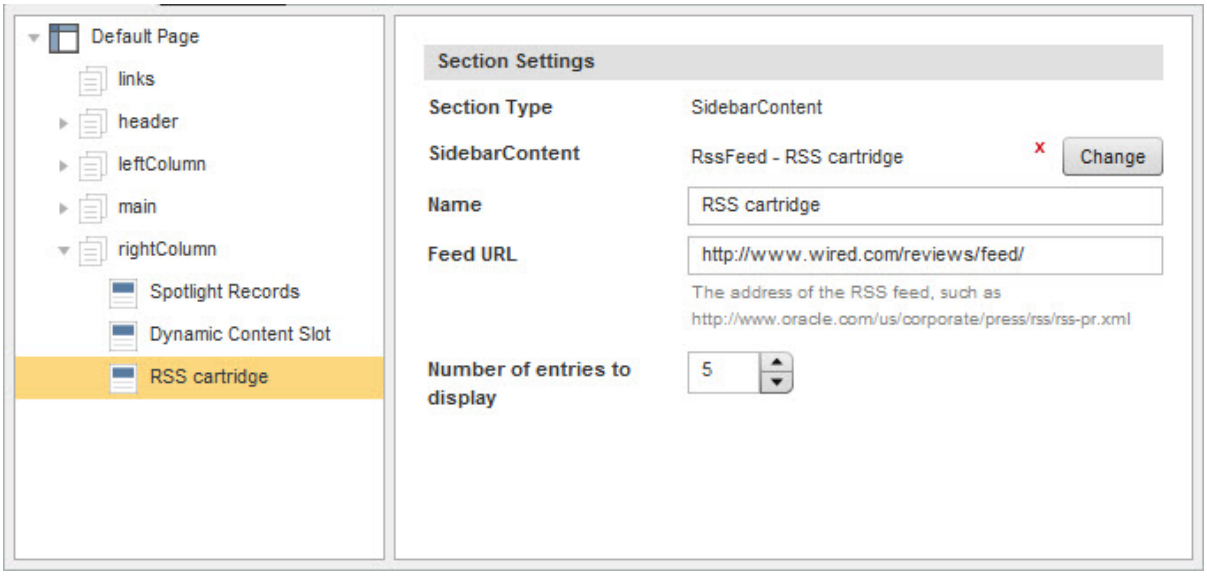
The business user needs to be able to configure the RSS Feed with a URL and the number of entries to display.

To create the RSS Feed template and add it to your application:

1. Create a new template based on the example below.

   Since the number of entries is expected to be an integer, the example uses a `NumericStepperEditor` for this property. It could also use a `SliderEditor` — both options guarantee that the value of the string property is an integer. In the example, we specify a default value of 5 for the number of entries.

2. Save the template with the name `SecondaryContent-RssFeed.xml` to the templates directory of your application.
3. Upload the template using the `set_templates` script.
4. Add the cartridge to the default search and navigation page as in the example below.



Note: The sample renderer for this cartridge works best with RSS feeds that have brief descriptions with no images or advertisements in the description field. A possible enhancement to this cartridge would be to make displaying the description configurable.

5. Save your changes to the page.

The cartridge instance configuration is saved as XML. At this point, because there is no cartridge handler specified for this cartridge, the same configuration is passed to the client as the response from the Assembler.

```
<ContentItem type="SecondaryContent">
  <TemplateId>RssFeed</TemplateId>
  <Name>RSS cartridge</Name>
  <Property name="feedUrl">
    <String>http://www.wired.com/reviews/feed/</String>
  </Property>
  <Property name="numEntries">
    <String>5</String>
  </Property>
</ContentItem>
```

The following shows the sample template for the RSS Feed cartridge:

```
<ContentTemplate xmlns="http://endeca.com/schema/content-template/2008"
                 xmlns:editors="editors"
                 type="SecondaryContent"
                 id="RssFeed">
  <Description>A cartridge that displays entries from an RSS feed.</Descrip¬
tion>
  <ThumbnailUrl>/ifcr/tools/xmgr/img/template_thumbnails/sidebar_con¬
tent.jpg</ThumbnailUrl>
  <ContentItem>
    <Name>RSS cartridge</Name>
    <Property name="feedUrl">
      <String/>
    </Property>
    <Property name="numEntries">
      <String>5</String>
    </Property>
  </ContentItem>
  <EditorPanel>
    <BasicContentItemEditor>
      <editors:StringEditor propertyName="feedUrl" label="Feed URL"
        bottomLabel="The address of the RSS feed, such as http://www.ora¬
cle.com/us/corporate/press/rss/rss-pr.xml"/>
      <editors:NumericStepperEditor propertyName="numEntries"
        label="Number of entries to display" minValue="1" maxValue="15"/>
    </BasicContentItemEditor>
  </EditorPanel>
</ContentTemplate>
```

Creating the cartridge handler

The cartridge handler fetches the RSS results and returns an output model to the client suitable for rendering.

To create the RSS Feed cartridge handler and add it to the application:

1. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `AbstractCartridgeHandler`.
2. Compile the cartridge handler and add the compiled class to your application.
3. Configure the Assembler to use the `RssFeedHandler` for the RSS Feed cartridge by adding the following to the Spring context file:

   ```
   <!--
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ~ BEAN: CartridgeHandler_RssFeed
   -->
   <bean id="CartridgeHandler_RssFeed"
   class="com.endeca.sample.cartridges.RssFeedHandler"
   scope="prototype" />
   ```

4. Restart the Endeca Tools Service.
5. Refresh the application.

The RSS feed does not display yet because we have not created the renderer, but you can validate that the response model has been populated with the information that we want to display via the JSON view:

```
{

    "@type": "RssFeed",
    "name": "RSS cartridge",
    "feedUrl": "http://www.wired.com/reviews/feed/",
    "numEntries": "5",
    "chanTitle": "Product Reviews",
    "chanUrl": "http://www.wired.com/reviews",
    "entries": [
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        },
        {
            "@type": "rssEntry",
            "itemDesc": "(description text omitted from this example)",
            "itemTitle": "(title text omitted from this example)",
            "itemUrl": "(url omitted from this example)"
        }
    ]

}
```

The following shows the code for the sample RSS Feed cartridge handler:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;
import com.endeca.infront.assembler.BasicContentItem;
import java.net.URL;
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.CharacterData;
```

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

public class RssFeedHandler extends AbstractCartridgeHandler {

 public ContentItem process(ContentItem pContentItem)
   throws CartridgeHandlerException {

  final String urlString = (String) pContentItem.get("feedUrl");
  final int numEntries =
    Integer.parseInt((String)pContentItem.get("numEntries"));

  try {
   URL url = new URL(urlString);
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   DocumentBuilder docBuilder = factory.newDocumentBuilder();
   Document RssContents = docBuilder.parse(url.openStream());

   // get the channel info
   Element channel =
    (Element)RssContents.getElementsByTagName("channel").item(0);
   pContentItem.put("chanTitle", getElementValue(channel, "title"));
   pContentItem.put("chanUrl", getElementValue(channel, "link"));

   // get the entries and add them to a list
   ArrayList<ContentItem> entries = new ArrayList<ContentItem>(numEntries);

   NodeList nodes = RssContents.getElementsByTagName("item");
   for(int i=0; i<numEntries; i++) {
    Element element = (Element)nodes.item(i);
    if (element!=null) {
     ContentItem entry = new BasicContentItem("rssEntry");
     entry.put("itemTitle", getElementValue(element, "title"));
     entry.put("itemUrl", getElementValue(element, "link"));
     entry.put("itemDesc", getElementValue(element, "description"));
     entries.add(entry);
    }
   }
   pContentItem.put("entries", entries);
  }
  catch (Exception e) {
   throw new CartridgeHandlerException(e);
  }

  return pContentItem;
 }

 private static String getCharacterDataFromElement(Element e) {
  try {
   Node child = e.getFirstChild();
   if(child instanceof CharacterData) {
    CharacterData cd = (CharacterData) child;
    return cd.getData();
   }
  }
  catch(Exception ex) {
  }
  return "";
```

```
 }

 private static String getElementValue(Element parent, String label) {
  return getCharacterDataFromElement(
    (Element)parent.getElementsByTagName(label).item(0));
 }

}
```

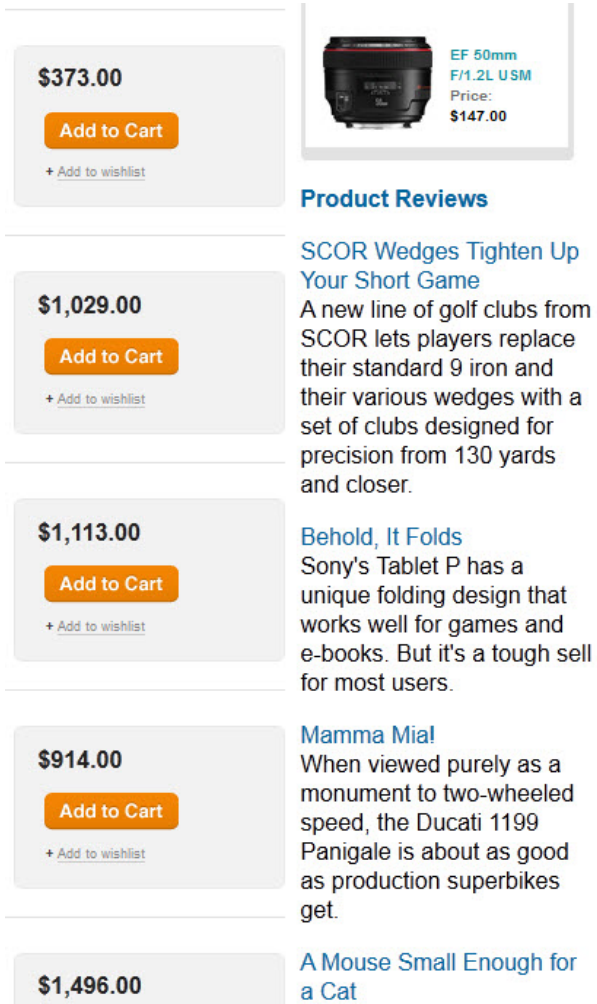
Creating the cartridge renderer

The renderer displays a summary of the results with links that take the site visitor to the site that originated the RSS feed.

To create a renderer for the RSS feed:

1. Create a new JSP page based on the example below.
2. Save the renderer to `/WEB-INF/views/desktop/RssFeed/RssFeed.jsp`.
3. Refresh the application to see the result.

The results from the RSS feed display in the right sidebar.



The following shows the code for the sample RSS Feed renderer in JSP:

```
<%@page language="java" pageEncoding="UTF-8"
contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>

<div style="padding:2ex 0">
<b><a href="${component.chanUrl}">${component.chanTitle}</a></b>
<c:forEach var="rssEntry" items="${component.entries}">
    <p><a href="${rssEntry.itemUrl}">${rssEntry.itemTitle}</a><br/>
    ${rssEntry.itemDesc}</p>
</c:forEach>
</div>
```

Custom Record Details cartridge with availability information

In this example, we extend the Record Details cartridge to display information about the availability of a product.

It demonstrates the following use cases:

- Extending one of the core Endeca cartridges
- Combining results from the MDEX Engine with information from another source during the `process()` phase of the assembly cycle
- Configuring a third-party service through Spring

In this cartridge, we create the following components:

Component	Description
cartridge handler	Extends the `RecordDetailsHandler` to add a property to the response model containing availability information.
mock "availability service"	Stands in for a real source of availability information such as an inventory system.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge. To enable the full functionality of this cartridge, the renderer should be updated to display the availability information, however that is not demonstrated in this guide.

Creating the cartridge handler and supporting classes

The `AvailabilityRecordDetailsHandler` extends the core `RecordDetailsHandler` to call a simple mock availability service to retrieve availability information about a particular record.

To create a cartridge handler that calls an availability service:

1. Create the following classes: `Availability`, `AvailabilityService`, and `FixedAvailabilityService` based on the examples below.

The `AvailabilityService` defines an interface that returns availability information based on a record identifier, and `FixedAvailabilityService` provides a basic implementation of the interface.

2. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `RecordDetailsHandler`.

   The handler takes the results of the MDEX Engine query and adds an additional property that represents the product availability.

3. Compile the classes and add them to your application.
4. Configure the Assembler to use the `AvailabilityRecordDetailsHandler` for the Record Details cartridge by editing the Spring context file as in the following example.

Note: If you have created a `sample-cartridge-config.xml` file for configuring the example cartridges, copy the `CartridgeHandler_ResultsList` bean from `assembler-context.xml` to your sample context file, comment out the version in `assembler-context.xml`, and then modify the version in your sample context file as indicated below.

```
<bean id="CartridgeHandler_RecordDetails"
  class="com.endeca.sample.cartridges.AvailabilityRecordDetailsHandler"
  parent="NavigationCartridgeHandler" scope="prototype" >
    <property name="recordState" ref="recordState" />
    <property name="availabilityService" ref="availabilityService" />
    <property name="recordSpec" value="common.id" />
    <property name="availabilityPropertyName"
      value="product.availability" />
</bean>

<bean id="availabilityService"
  class="com.endeca.sample.cartridges.FixedAvailabilityService"
  scope="singleton" >
  <!-- Implementation-specific configuration for the service
       could be specified here -->
</bean>
```

5. Restart the Endeca Tools Service.
6. Refresh the application and then click on any record to view its details page.

The availability property is now returned as part of the record details information:

```
{
    "@type": "RecordDetailsPageSlot",
    "name": "Record Details Page",
    "contentCollection": "Record Details Pages",
    "ruleLimit": "1",
    "contents": [
        {
            …
            },
            "recordDetails": {
                "@type": "RecordDetails",
                "record": {
                   "@class": "com.endeca.infront.cartridge.model.Record",
                   "numRecords": 1,
                   "attributes": {
                        …
                        "product.availability": [
                            "BACKORDER"
```

```
                        ],
                        …
                    },
                    "records": [ … ]
                },
                "name": "Record Details"
            }
        }
    ],
    …
}
```

The renderer can now be updated to display availability information based on the value of this property.

The following shows the code for the availability service and its supporting classes:

```
package com.endeca.sample.cartridges;

public enum Availability {
    IMMEDIATE,
    WEEK,
    DROP_SHIP,
    BACKORDER;

}
```

```
package com.endeca.sample.cartridges;

public interface AvailabilityService {

    Availability getAvailabilityFor(String identifier);
}
```

```
package com.endeca.sample.cartridges;

public class FixedAvailabilityService implements AvailabilityService {

    public Availability getAvailabilityFor(String identifier) {
        try {
            return Availability.valueOf(identifier);
        } catch (IllegalArgumentException e) {
            return Availability.BACKORDER;
        }
    }
}
```

The following shows the code for the custom cartridge handler:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.cartridge.RecordDetails;
import com.endeca.infront.cartridge.RecordDetailsConfig;
import com.endeca.infront.cartridge.RecordDetailsHandler;
import com.endeca.infront.cartridge.model.Attribute;
import org.springframework.beans.factory.annotation.Required;

public class AvailabilityRecordDetailsHandler extends RecordDetailsHandler
 {

    private AvailabilityService availabilityService;
    private String recordSpec;
```

```
    private String availabilityPropertyName;

    @Required
    public void setAvailabilityService(
      AvailabilityService availabilityService_) {
        availabilityService = availabilityService_;
    }

    @Required
    public void setRecordSpec(String recordSpec_) {
        recordSpec = recordSpec_;
    }

    @Required
    public void setAvailabilityPropertyName(
      String availabilityPropertyName_) {
        availabilityPropertyName = availabilityPropertyName_;
    }

    @Override
    public RecordDetails process(RecordDetailsConfig detailsConfig)
      throws CartridgeHandlerException {
        RecordDetails details = super.process(detailsConfig);
        if (null == details) return null;
        Attribute attr =
          details.getRecord().getAttributes().get(recordSpec);
        if (null == attr || 1 != attr.size()) {
            throw new CartridgeHandlerException("No record spec
              available on record, or spec is multiassign");
        }
        Attribute<Availability> availability =
          new Attribute<Availability>();
        availability.add(
          availabilityService.getAvailabilityFor(attr.toString()));
        details.getRecord().getAttributes().put(availabilityPropertyName,
          availability);
        return details;
    }

}
```

Custom Results List with recommendations

In this example, we extend the Results List cartridge to boost certain products based on information from a recommendation engine.

It demonstrates the following use cases:

- Extending one of the core Endeca cartridges
- Using data from another source to modify the query to the MDEX Engine created during the `pre¬process()` phase of the assembly cycle
- Configuring a third-party service through Spring

In this cartridge, we create the following components:

Component	Description
cartridge handler	Extends the `ResultsListHandler` to retrieve a set of items to boost from a recommendations engine and add a boost stratum to the MDEX Engine query.
mock recommendations service	Stands in for a real source of recommendations.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge. Additionally, the response model for the customized cartridge is the same as the default Results List (only with the records in a different order), so there is no need for changes to the default renderer.

Creating the cartridge handler and supporting classes

The `RecommendationsResultsListHandler` extends the core `ResultsListHandler` to call a simple mock recommendations service and boosts the recommended products.

To create a cartridge handler that boosts recommended records:

1. Create the interface `RecommendationService` and the concrete implementation `TestRecommendationService` based on the examples below.

   As a proof of concept, the recommendations service always returns the same recommendations from the Discover Electronics data set.

2. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `ResultsListHandler`.

   The handler retrieves a list of recommended records from the service and adds them to a boost stratum for the MDEX Engine query. If the records are present in the results set, they are boosted to the top of the results list.

3. Compile the classes and add them to your application.
4. Configure the Assembler to use the `RecommendationsResultsListHandler` for the Results List cartridge by editing the Spring context file as follows:

   Note: If you have created a `sample-cartridge-config.xml` file for configuring the example cartridges, copy the `CartridgeHandler_ResultsList` bean from `assembler-context.xml` to your sample context file, comment out the version in `assembler-context.xml`, and then modify the version in your sample context file as indicated below.

```
<bean id="CartridgeHandler_ResultsList"
  class="com.endeca.sample.cartridges.RecommendationsResultsListHandler"

  parent="NavigationCartridgeHandler" scope="prototype">
    <property name="contentItemInitializer">
      <!-- additional elements omitted from this example -->
    </property>
    <property name="sortOptions">
      <!-- additional elements omitted from this example -->
    </property>
    <property name="recommendationService" ref="recommendationService"
/>
    <property name="recordSpec" value="common.id"/>
</bean>
```

```
<bean id="recommendationService"
  class="com.endeca.sample.cartridges.TestRecommendationService"
  scope="singleton" >
  <!-- Implementation-specific configuration for the service
       could be specified here -->
</bean>
```

5. Restart the Endeca Tools Service.
6. Refresh the application.

The recommended records are boosted to the top of the results:

Display: 10 per page Sort By: Relevance

Canon
PowerShot S95

PowerShot S95, 10MP, 3.8x Optical. 4x Digital. 15x Combined Zoom, 7.62 cm (3.0 ") LCD, 3648 x 2048, 16:9, Zwart

$1,000.00
Add to Cart
+ Add to wishlist

Trust
Cuby Pro

Cuby Pro, Titanium, 1.3 Mp, 1280 x 1024, USB 2.0

$918.00
Add to Cart
+ Add to wishlist

Pentax
K20D

K20D, 14.6 Megapixels

$907.00
Add to Cart
+ Add to wishlist

Fujifilm
FinePix F50fd & SD Card 1GB

FinePix F50fd Black & SD Card 1GB

$991.00
Add to Cart
+ Add to wishlist

Kodak
EasyShare M863

1319094, EasyShare M863 Digital Camera, 8.2 MP for prints up to 30 × 40 in. (76 × 102 cm), 3X optical zoom

$964.00
Add to Cart
+ Add to wishlist

Pentax
Optio M40

Optio M40 Digital Camera

$747.00
Add to Cart
+ Add to wishlist

Sony
DSC-H50

9.1 Mega Pixel Cyber-shot H50 Series Digital Camera

$991.00
Add to Cart
+ Add to wishlist

The following shows the code for the recommendations service interface and concrete implementation:

```
package com.endeca.sample.cartridges;

import java.util.List;

public interface RecommendationService {
    public List<String> getRecommendedProductIds();
}
```

```
package com.endeca.sample.cartridges;

import java.util.Arrays;
import java.util.List;

public class TestRecommendationService
  implements RecommendationService {
    public static final List<String> IDS =
      Arrays.asList("5891932", "6001963", "1438066", "1581692",
        "2708142", "1235424", "3422480");

    public List<String> getRecommendedProductIds() {
        return IDS;
    }

}
```

The following shows the code for the custom cartridge handler:

```
package com.endeca.sample.cartridges;

import java.util.ArrayList;
import java.util.List;

import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.cartridge.ResultsListConfig;
import com.endeca.infront.cartridge.ResultsListHandler;
import com.endeca.infront.navigation.model.CollectionFilter;
import com.endeca.infront.navigation.model.PropertyFilter;

public class RecommendationsResultsListHandler extends ResultsListHandler
 {
    private RecommendationService recommendationService;
    private String recordSpec;

    public String getRecordSpec() {
        return recordSpec;
    }

    public void setRecordSpec(String recordSpec_) {
        this.recordSpec = recordSpec_;
    }

    public void setRecommendationService(
      RecommendationService recommendationService_) {
        recommendationService = recommendationService_;
    }

    /**
     * This cartridge will get the list of recommended products
     * (by record spec) and explicitly boost each one of them using
```

```
     * a PropertyFilter.
     */
    @Override
    public void preprocess(ResultsListConfig pContentItem)
      throws CartridgeHandlerException {
        List<String> ids =
          recommendationService.getRecommendedProductIds();
        List<CollectionFilter> boostFilters =
          new ArrayList<CollectionFilter>(
                ids.size());
        for (String s : ids) {
            boostFilters.add(new CollectionFilter(new PropertyFilter(
                    recordSpec, s)));
        }

        pContentItem.setBoostStrata(boostFilters);
        super.preprocess(pContentItem);
    }

}
```

"Hello, World" cartridge with layered color configuration

In this example, we extend the "Hello, World" example cartridge to demonstrate the layering of configuration from several sources.

In this scenario, we can define a default color for the message in our "Hello, World" cartridge, which the business user can override on a per-instance basis in Experience Manager. The site visitor can also select a preferred color from the client application.

It demonstrates the following use cases:

- Combining the default cartridge configuration, cartridge instance configuration, and request-based configuration using the `ConfigInitializer` and `RequestParamMarshaller` helper classes
- Using a cartridge configuration bean

In this cartridge, we create the following components:

Component	Description
cartridge handler	Uses the `ColorConfigInitializer` to layer multiple sources of configuration for message color.
cartridge configuration bean	Provides a means of specifying default values for this cartridge via Spring.
cartridge renderer	Provides a drop-down list from which the site visitor can choose a color for the message.

Because this cartridge does not introduce any change in the configuration options for the business user, there are no template changes for this cartridge.

Creating the cartridge handler and supporting classes

The cartridge handler combines the various sources of configuration for message color using the `ConfigInitializer` and `RequestParamMarshaller` helper classes.

To create the "Hello, World" cartridge handler with color configuration and add it to the application:

1. Create a new Java class in the package `com.endeca.sample.cartridges` based on the example below, which extends `AbstractCartridgeHandler`.
2. Create a configuration bean for this cartridge based on the example below. This enables us to define default values for the cartridge properties in the Spring context file.
3. Compile the cartridge handler and configuration bean and add them to your application.
4. Configure the Assembler to use the `ColorConfigHandler` for the "Hello, World" cartridge by editing the Spring context file as follows:

```
<bean id="CartridgeHandler_Hello"
  class="com.endeca.sample.cartridges.ColorConfigHandler"
  scope="prototype">
  <property name="contentItemInitializer">
    <bean class="com.endeca.infront.cartridge.ConfigInitializer"
      scope="singleton">
      <property name="defaults">
        <bean class="com.endeca.sample.cartridges.ColorConfig"
          scope="singleton">
          <property name="messageColor" value="#FF6600"/>
        </bean>
      </property>
      <property name="requestParamMarshaller">
        <bean
          class="com.endeca.infront.cartridge.RequestParamMarshaller"
          scope="singleton">
         <property name="httpServletRequest" ref="httpServletRequest"/>

          <property name="requestMap">
            <map>
              <entry key="color" value="messageColor"/>
            </map>
          </property>
        </bean>
      </property>
    </bean>
  </property>
  <property name="colorOptions">
    <map>
      <entry key="Red" value="#FF0000"/>
      <entry key="Green" value="#00FF00"/>
      <entry key="Blue" value="#0000FF"/>
      <entry key="Black" value="#000000"/>
    </map>
  </property>
</bean>
```

5. Restart the Endeca Tools Service.
6. Refresh the application.

The color options do not display yet because we have not updated the renderer, but you can validate that the response model has been populated with the information that we want the renderer to use via the JSON view:

```
{

    "@type": "Hello",
    "name": "Hello cartridge",
    "message": "Hello, color world!",
    "messageColor": "#0000FF",
    "colorOptions": [
        {
            "@type": "colorOption",
            "hexCode": "#FF0000",
            "label": "Red"
        },
        {
            "@type": "colorOption",
            "hexCode": "#00FF00",
            "label": "Green"
        },
        {
            "@type": "colorOption",
            "hexCode": "#0000FF",
            "label": "Blue"
        },
        {
            "@type": "colorOption",
            "hexCode": "#000000",
            "label": "Black"
        }
    ]

}
```

The following shows the code for the sample "Hello, World" cartridge handler with color configuration:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.AbstractCartridgeHandler;
import com.endeca.infront.assembler.CartridgeHandlerException;
import com.endeca.infront.assembler.ContentItem;
import com.endeca.infront.assembler.BasicContentItem;
import com.endeca.infront.assembler.ContentItemInitializer;
import com.endeca.sample.cartridges.ColorConfig;
import java.util.ArrayList;
import java.util.Map;

public class ColorConfigHandler extends AbstractCartridgeHandler {

 private ContentItemInitializer mInitializer;
 private Map<String, String> mColorOptions;

  public void setContentItemInitializer(ContentItemInitializer initializer)
 {
    mInitializer = initializer;
  }

  public void setColorOptions(Map<String, String> colorOptions) {
    mColorOptions = colorOptions;
```

```
  }

  /**
   * Returns the merged configuration based on Spring defaults,
   * Experience Manager configuration, and request parameters
   */
  @Override
  public ContentItem initialize(ContentItem pContentItem) {
    // If any configuration from Experience Manager is empty, remove
    // that property so we can use the default value
    for (String key: pContentItem.keySet()) {
      if (((String)pContentItem.get(key)).isEmpty())
        pContentItem.remove(key);
    }
      return mInitializer == null ? new ColorConfig(pContentItem) :
        mInitializer.initialize(pContentItem);
    }

  /**
   * Returns the merged configuration and information about the color op¬
tions
   * available to the site visitor.
   */
  @Override
  public ContentItem process(ContentItem pContentItem)
    throws CartridgeHandlerException {
    int numColors = mColorOptions.size();
    ArrayList<ContentItem> colors =
      new ArrayList<ContentItem>(numColors);
    if (mColorOptions != null && !mColorOptions.isEmpty()) {
      for (String key: mColorOptions.keySet()) {
        ContentItem color = new BasicContentItem("colorOption");
        color.put("label", key);
        color.put("hexCode", mColorOptions.get(key));
        colors.add(color);
      }
      pContentItem.put("colorOptions", colors);
    }
    return pContentItem;
  }
}
```

The following code implements a basic bean that enables us to specify a default value for the message color in the Spring configuration:

```
package com.endeca.sample.cartridges;

import com.endeca.infront.assembler.BasicContentItem;
import com.endeca.infront.assembler.ContentItem;

public class ColorConfig extends BasicContentItem {

  public ColorConfig() {
    super();
  }

  public ColorConfig(final String pType) {
    super(pType);
  }

  public ColorConfig(ContentItem pContentItem) {
```

```
    super(pContentItem);
  }

  public String getMessageColor() {
    return getTypedProperty("messageColor");
  }

  public void setMessageColor(String color) {
    this.put("messageColor", color);
  }
}
```

Creating the cartridge renderer

In this example we update the "Hello, World" renderer to add a control for the site visitor to select a color for the message.

To add a drop-down for the site visitor to select a message color based on the options configured for this cartridge:

1. Create a new JSP page based on the example below, or update the renderer you previously created by adding the section in bold.
2. Save the renderer to `/WEB-INF/views/desktop/Hello/Hello.jsp`.
3. Refresh the application to verify that the drop-down menu displays.

The following shows the code for the sample "Hello, World" renderer with color choice drop-down in JSP:

```
<%@page language="java" pageEncoding="UTF-8"
contentType="text/html;charset=UTF-8"%>

<%@include file="/WEB-INF/views/include.jsp"%>
<div style="border-style: dotted; border-width: 1px;
border-color: #999999; padding: 10px 10px">
  <div style="font-size: 150%;
    color: ${component.messageColor}">${component.message}
  </div>
  <div style="font-size: 80%; padding: 5px 0px">
    <select onchange="location = this.options[this.selectedIndex].value">
      <option value="">Select a color</option>
      <c:forEach var="colorOption" items="${component.colorOptions}">
        <c:url value="<%= request.getPathInfo() %>" var="colorAction">
          <c:param name="color" value="${colorOption.hexCode}" />
        </c:url>
        <option value="${colorAction}">${colorOption.label}</option>
      </c:forEach>
    </select>
```

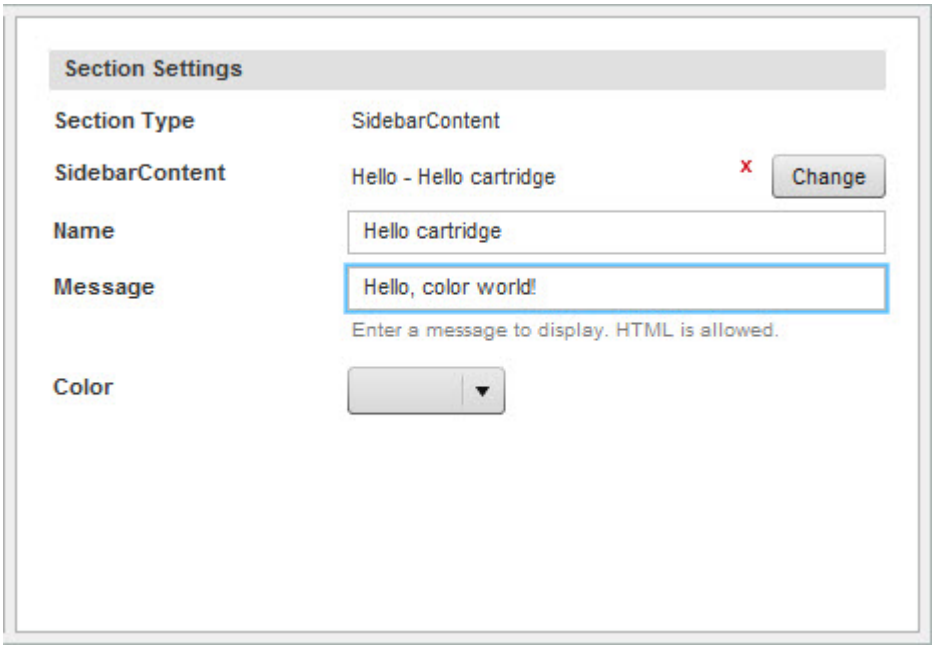
```
  </div>
</div>
```

Testing the "Hello, World" cartridge with layered color configuration

We can validate that the cartridge handler applies the different sources of configuration properly by incrementally populating each source of the configuration.

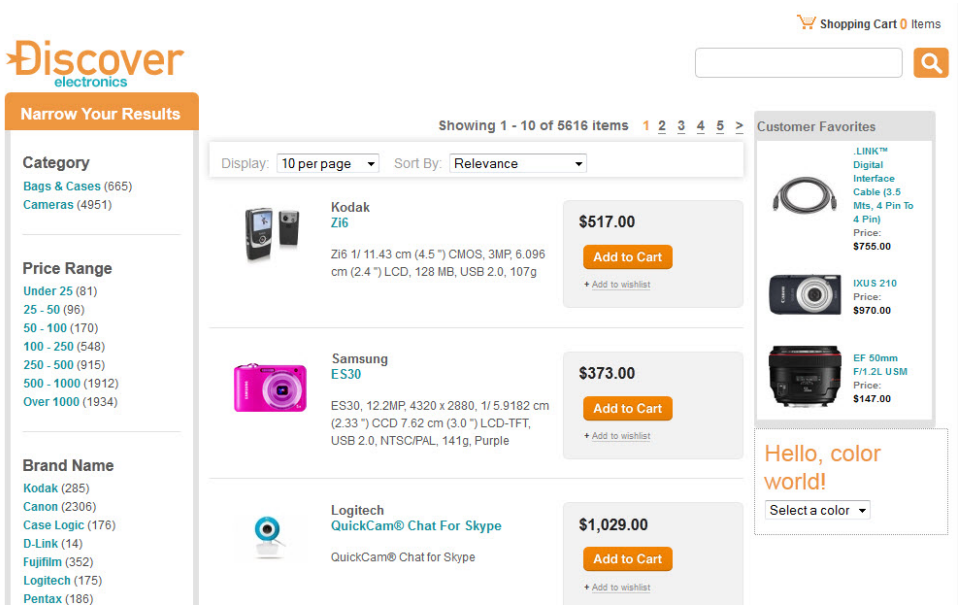
To test the "Hello, World" cartridge:

1. In Experience Manager, remove any previously created instance of the Hello cartridge.
2. Insert a new instance of the cartridge on the default page and specify a message string, but do not select a color.

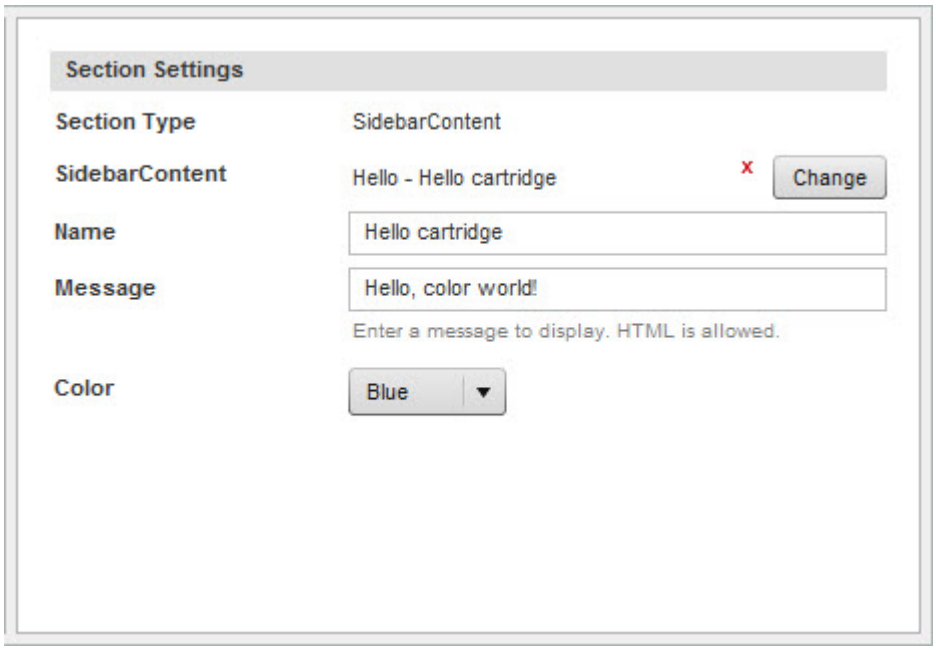


3. Save the page.
4. Refresh the application.

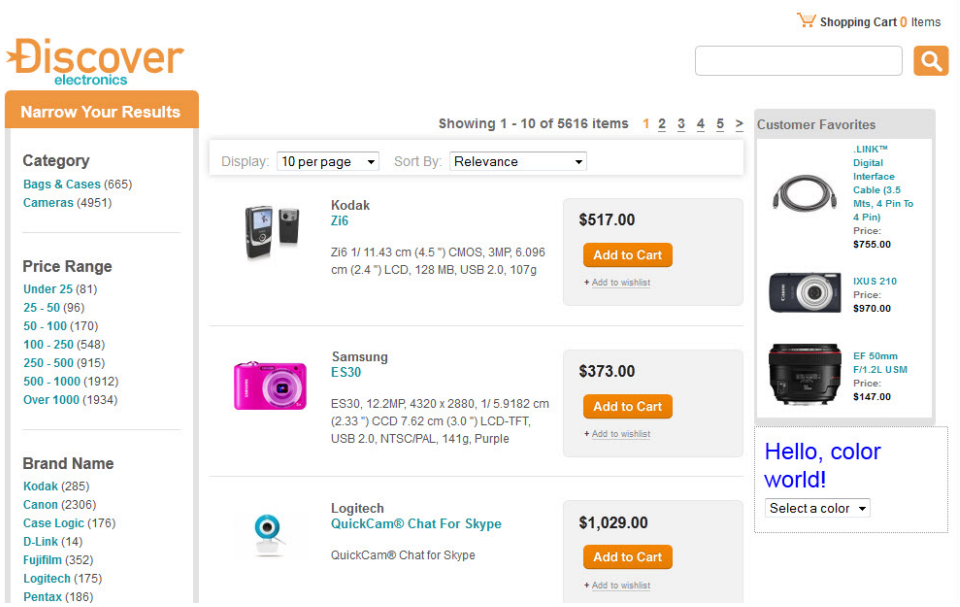
   The message displays using the default color, orange.



5. Going back to Experience Manager, now select a message color for this instance of the cartridge.

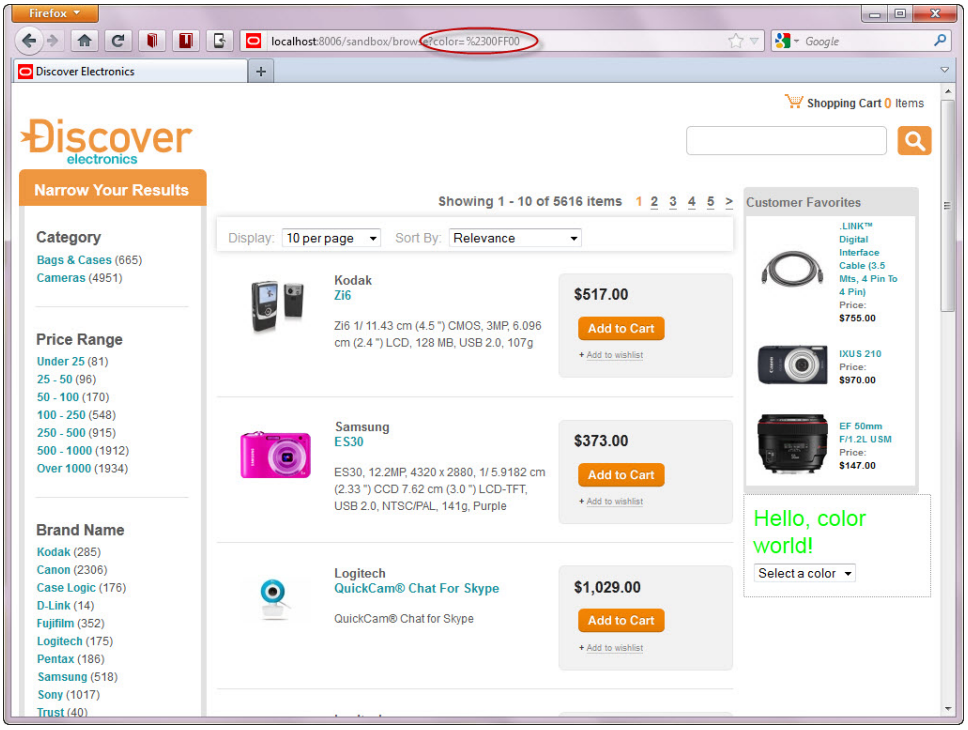


6. Refresh the application.
   The message displays using the color configured in Experience Manager.



7. Using the drop-down list on the cartridge, select another color.

   The drop-down control adds a `color` parameter to the URL, which is parsed by the `RequestParam¬Marshaller` into the `messageColor` property.



Index